A formalization of multi-tape Turing machines

A Formalization of Multi-tape Turing Machines

Andrea Asperti[a], Wilmer Ricciotti[b,1]

[a] *Department of Computer Science and Engineering, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy*

[b] *IRIT, University of Toulouse, 118 Route de Narbonne, 31062 Toulouse Cedex 9, France*

Abstract

We discuss the formalization, in the Matita Theorem Prover, of basic results on multi-tapes Turing machines, up to the existence of a (certified) Universal Machine, and propose it as a natural benchmark for comparing different interactive provers and assessing the state of the art in the mechanization of formal reasoning. The work is meant to be a preliminary step towards the creation of a formal repository in Complexity Theory, and is a small piece in our long-term Reverse Complexity program, aiming to a comfortable, machine independent axiomatization of the field.

1. Introduction

In spite of the remarkable achievements recently obtained in the automatic checking of complex results in many different domains, spanning from pure mathematics [13, 23, 7, 21] to software verification [28, 27, 36, 1], passing through the metatheory and semantics of programming languages [15, 33], the learning curve of interactive provers is still extremely steep and the mechanization of formal reasoning remains a very complex task.

Even comparing proof assistants in terms of their concrete usability [34, 6] looks difficult, due to the different domains of applications (for instance, alpha equivalence and binding mechanisms are important issues for the metatheory of functional languages and type systems [33], but are almost negligible for proving properties on compilers or other software tools [2]) and the different criteria that can be used in the comparison (in [14], it is particularly stressed the importance of having human readable proofs, while [2] shifts the emphasis on the connection between the proof of a specification and the corresponding implementation, and the possibility of extracting or automatically checking it).

Email addresses: asperti@cs.unibo.it (Andrea Asperti), wilmer.ricciotti@irit.fr (Wilmer Ricciotti)

[1] Wilmer Ricciotti is supported by the project AJITPROP of the Fondation Airbus.

Creating several benchmarks in different domains seems to be the best approach for promoting the actual development of the field and for comparing theorem provers on their ability to express/synthesize proofs and to support the user in the process of formalization.

Somewhat surprisingly, very little work has been done so far in major fields of theoretical computer science, such as computability theory and, especially, complexity theory. Two notable exceptions are the articles by Norrish [29] and, very recently, by Xu, Zhang and Urban [35]. Norrish's work contains basic results in computability theory relying on λ-calculus and recursive functions as computational models. A major drawback of this approach is that computational constructs like β-reduction or recursive function call are not finitistic and hence are not suitable for a foundational investigation of complexity. In spite of the many attempts of providing machines independent characterizations of Complexity Theory, its actual foundation still relies on Turing machines.

Turing machines are the actual focus of Xu, Zhang and Urban's work [35], that testifies the new, recent interest of the interactive theorem prover community on these topics. Their contribution appeared after the first version of this article was submitted, and our development was completely independent and uninfluenced by their work. We shall postpone the comparison with [35] to the conclusions.

As a matter of fact, the subject of Turing machines provides an excellent test bench for proof assistants, for several reasons

1. it is a well established subject, not deserving auxiliary specifications; even undergraduate students can directly tackle it, possibly as an experimental part of traditional theoretical courses;
2. it does not require sophisticated mathematics; it can be easily formalized in first order logic, but it can also take advantage of more sophisticated logical frameworks, like systems with dependent types (for instance, we found both natural and convenient to use them to formalize multi-tape machine by means of vectors of a specified length);
3. it provides simple but effective examples of *executable* specifications, allowing to test the computational capabilities of the system and/or its code-extraction facilities;
4. it allows to address interesting termination problems, that is one of the most complex issue in the mechanization of formal reasoning.

In this paper, we present a down-to-earth formalization for the Matita Interactive Theorem Prover [9, 10] of the theory of multi-tape Turing machines, up to the existence of a universal machine and the proof of its correctness. The code is available at `http://www.cs.unibo.it/~asperti/turing.tar`.

In our development, we have been inspired by several traditional articles and textbooks, comprising e.g. [20, 26, 18, 32, 30]; all of them are excellent works, and provide a rigorous mathematical exposition of the theory of Turing machines, and nevertheless none of them gives a sufficiently accurate description (especially of the universal machine) to be directly translated in formal terms.

Although we do not claim any particular originality in our approach, there are several points in the development where the choice of the data structures used for the formal encoding, or the means used for expressing the semantics of Turing machines have not been entirely straightforward. In particular, for obvious reasons, we tried to be *as compositional as possible* in the semantic specification, allowing us to discover unexpected compositional properties in a domain that is usually reputed to be (quite) non-compositional.

The current work is a major revisitation of the work presented at WoLLIC [8]; in particular, we switched from mono to multi-tape Machines, forcing us to completely rewrite the universal machine. While [8] was essentially a preliminary report on an ongoing effort, this paper describes a much more mature and stabilized version of the formalization. Relevant differences between the two works are emphasized at suitable places along the paper, and an overall comparison is given in the conclusions.

The structure of the paper is the following: Section 2 contains a short description of the main libraries of Matita used in the development; Section 3 gives the formal definition of multi-tape Turing machines and their semantics; Section 4 provides means for composing machines (sequential composition, conditionals and iteration); Section 5 contains the definition of basic, atomic machines and a few examples of slightly more complex machines derived from them by compositional mechanisms; Section 6 introduces the notion of Normal Turing machine and its standard representation as a list of tuples; Section 7 gives an outline of the universal machine; Sections 8,9 and 10 are respectively devoted to the three main routines of the universal machine, namely *finding* the right tuple to apply, and use it to *update* the state and *execute* the suitable action on the simulated tape; in Section 11, we summarize the main results which have been proved about the universal machine. In the conclusion we provide overall information about the size of the contribution and the resources required for its development as well as more motivations for pursuing formalization in computability and complexity theory: in particular we shall briefly outline our long term *Reverse Complexity* program, aiming to a trusted and machine independent axiomatization of the field, suitable for mechanization.

2. Preliminaries

Our formalization of Turing machines relies on two main libraries of Matita, that we shall rapidly discuss in this section: finite sets; and vectors. A theory of finite sets is essential not only to formalize data structures like the tape alphabet or the machine states, but also to transform a (executable!) transition function into a finite graph (that is, up to trivial encodings, the representation exploited by the universal machine). Vectors can be conveniently used to develop the theory of multi-tape machines in a fully parametric way.

2.1. Implicit Parameters

Before addressing Matita code, it is worth to spend a few words about the syntax for *implicit parameters*. Matita is a type-theory based interactive prover,

and proof checking is reduced to type-checking, via the well known Curry-Howard (Types as Propositions) analogy. Types are often redundant and can be automatically guessed by a suitable type inference algorithm [12]. The user may ask the system to automatically synthesize a given argument by replacing it by a question mark. Each question mark stands for a different parameter; a sequence of n questions marks stands for n implicit parameters (question marks do not need to be spaced). Instead of question marks, the user can also use ellipses (...), standing for an arbitrary number of implicit arguments: in this case we also delegate to the system the charge to figure out the correct number of arguments. Both question marks and ellipses could be made completely transparent by adding suitable notational extensions (a mechanism supported by Matita); however, the fact that a term expecting a given number of arguments is actually receiving in input only a subset of them can be sometimes quite confusing, and we prefer to explicitly leave a bit of notation to recall an omitted, implicit information.

2.2. Finite Sets

As customary in type-theoretic formalizations of abstract algebra, libraries are usually organized into a hierarchy of structures based on nested dependent record types. In Matita, at the bottom of this hierarchy there is the `DeqSet` structure, that deals with types equipped with a decidable equality (essentially analogous to the `eqType` in [22]):

```
record DeqSet : Type :=
{ carr :>Type;
   eqb: carr →carr →bool;
   eqb_true: ∀x,y. (eqb x y = true) ↔(x = y)}.
```

We recall that, in Matita, each definition of a record `foo` implicitly defines an associated *constructor* `mk_foo` that takes a list of arguments corresponding to the record fields, and returns a new record. The names given by the user to the record fields are associated to *projection functions*, extracting from the record the corresponding field value.

The `:>` symbol declares `carr` as a coercion from a DeqSet to its carrier type. Suppose we build a record `DeqNat:DeqSet` having the natural numbers `nat` as carrier. Then, for instance, the expression `0:DeqNat` is well typed, and it is understood by the system as `0:carr DeqNat`. We use the infix notation "==" for the decidable equality `eqb` of the set (different from the primitive logical equality of the system, denoted with "=").

Inductive data structures can usually be turned into a `DeqSet` by defining "==" as a structural equality test. This is for instance the case for booleans and natural numbers (and many others), for which we have two corresponding `DeqSets` named `DeqBool` and `DeqNat`. We use the mechanism of unification hints [11] to suggest to Matita that, for example, if it meets a natural number, and if required by unification, it may lift it up to an element of a `DeqSet` using a suitable decidable equality hinted to by the user. For instance, the expression `0 == 1` is well typed: the comparison would require two elements of type `carr ?X` for some

unknown DeqSet `?X`, while it receives two elements of type `nat`. The purpose of the hint is to suggest a possible solution for the equation `carr ?X = Nat`^{2}, namely, in this case, `?X = DeqNat`.

A finite set is a record consisting of a DeqSet A, a list of elements of type A enumerating all the elements of the set, and the proofs that the enumeration does not contain repetitions and is complete (`memb` is the membership operation on lists, defined in the obvious way; the question mark is an example of an *implicit parameter* automatically filled in by the system).

```
record FinSet : :=
{ FinSetcarr:> DeqSet;
  enum: list FinSetcarr;
  enum_unique: uniqueb FinSetcarr enum = true;
  enum_complete : ∀x:FinSetcarr. memb ? x enum = true}.
```

The library provides many operations for building new `FinSet`s by composing existing ones: for example, if `A` and `B` have type `FinSet`, then `option A`, `A×B`, `A⨁B` are finite sets too. In all these cases, unification hints are used to suggest the *intended* finite set structure associated with them, that makes their use quite transparent to the user.

Among all constructions, the most interesting one for the purpose of this paper is the arrow type `A→B`. We may define the graph of `f:A→B`, as the set (sigma type) of all pairs $\langle a, b\rangle$ such that $f(a) = b$.

```
definition graph_of :=λA,B.λf:A→B. ∑p:A×B.f (\fst p) = \snd p.
```

In case the equality is decidable, we may effectively enumerate all elements of the graph by simply filtering from pairs in `A×B` those satisfying the test `λp.f (\fst p) == \snd p)`:

```
definition graph_enum :=λA,B:FinSet.λf:A→B.
  filter ? (λp.f (\fst p) == \snd p) (enum (FinProd A B)).
```

The proofs that this enumeration does not contain repetitions and is complete are straightforward.

2.3. Vectors

A vector of length n of elements of type A is simply defined in Matita as a record composed by a list of elements of type A, plus a proof that the list has the expected length. Vectors are a paradigmatic example of *dependent* type, that is of a type whose definition depends on a term.

```
record Vector (A:Type) (n:nat): Type :=
{ vec :>list A;
  len: length ? vec = n }.
```

[2]The solution to this type of unification problems would require narrowing, that is a complex and expensive operation transcending the (pseudo)-first order capabilities of the unification algorithms typically (and wittingly) implemented in proof assistants.

Given a list l we may trivially turn it into a vector of length $|l|$; we just need to prove that $|l| = |l|$ that follows from the reflexivity of equality.

```
definition Vector_of_list :=λA,l.mk_Vector A (|l|) l (refl ??).
```

Most functions operating on lists can be naturally extended to vectors: interesting cases are `vec_append`, concatenating vectors, and `vec_map`, mapping a function f on all elements of the input vector and returning a vector of the same dimension of the input one.

Since a vector is automatically coerced, if needed, to the list of its elements, we may simply use typical functions over lists (such as `nth`) to extract/filter specific components.

The function `change_vec A n v a i` replaces the content of the vector v at position i with the value a.

The most frequent operation on vectors for the purposes of our work is their comparison. In this case, we have essentially two possible approaches: we may proceed component-wise, using the following lemma

```
lemma eq_vec: ∀A,n.∀v1,v2:Vector A n.∀d.
  (∀i. i < n →nth i A v1 d = nth i A v2 d) →v1 = v2.
```

or alternatively we may manipulate vectors exploiting the commutation or idempotence of `change_vec` and its interplay with other operations.

3. The notion of Turing machine

Turing machines were defined by Alan M. Turing in [31]. To computer scientists, they are a very familiar notion, so we shall address straight away their formal definition. We formalized both mono and multi-tape machines[3]; in many cases, simple multi tape (sub-)machines work on a single tape at a time, and it is convenient to define them as "injections" of mono tapes machines.

3.1. The tape

A first interesting issue is the definition of the tape. The natural idea is to formalize it as a zipper, that is a pair of lists l and r, respectively representing the portions of the tape at the left and the right of the tape head; by convention, we may assume the head is reading the first symbol on the right. Of course, the machine must be aware this list can be empty, that means that the transition function should accept an *optional* tape symbol as input. Unfortunately, in this way, the machine is only able to properly react to a right overflow; the

[3]It is worth to recall that the choice about the number of tapes, while irrelevant for computability issues, it is not from the point of view of complexity. Hartmanis and Stearns [24] have shown that any k-tape machine can be simulated by a one-tape machine with at most a quadratic slow-down, and Hennie [25] proved that in some cases this is the best we can expect; Hennie and Stearns provided an efficient simulation of multi-tape machines on a two-tape machine with just a logarithmic slow-down [19].

problem arises when the left tape is empty and the head is moved to the left: a new "blank" symbol should be added to the right tape. A common solution in textbooks is to reserve a special blank character $\sqcup$ of the tape alphabet for this purpose: the annoying consequence is that tape equality should be defined only up to a suitable equivalence relation ignoring blanks. To make an example, suppose we move the head to the left and then back to the right: we expect the tape to end up in the same situation we started with. However, if the tape was in the configuration $([\,], r)$ we would end up in $([\sqcup], r)$. As anybody with some experience in interactive proving knows very well, reasoning up to equivalence relations is *extremely* annoying, that prompts us to look for a different representation of the tape.

The main source of our problem was the asymmetric management of the left and right tape, with the arbitrary assumption that the head symbol was part of the right tape. If we try to have a more symmetric representation we must clearly separate the head symbol from the left and right tape, leading to a configuration of the kind (l, c, r) (mid-tape); if we have no c, this may happen for three different reasons: we are on the left end of a non-empty tape (left overflow), we are on the right end of a non-empty tape (right overflow), or the tape is completely empty. This definition of the tape may seem conspicuous at first glance, but it resulted to be quite convenient in practice.

```
inductive tape (sig:FinSet) : Type :=
| niltape : tape sig
| leftof : sig →list sig →tape sig
| rightof : sig →list sig →tape sig
| midtape : list sig →sig →list sig →tape sig.
```

For instance, suppose to be in a configuration with an empty left tape, that is (*midtape* [] *a l*); moving to the left will result in (*leftof a l*); further moves to the left are forbidden, and moving back to the right restores the original situation.

Given a tape, we may easily define the optional current symbol

```
definition current :=λsig.λt:tape sig.match t with
 [ midtape _ c _ ⇒Some ? c | _ ⇒None ? ].
```

as well as the left and right portions of the tape:

```
definition left :=λsig.λt:tape sig.match t with
 [ niltape ⇒[] | leftof _ _ ⇒[] | rightof s l ⇒s::l | midtape l _ _ ⇒l ].

definition right :=λsig.λt:tape sig.match t with
 [ niltape ⇒[] | leftof s r ⇒s::r | rightof _ _ ⇒[] | midtape _ _ r ⇒r ].
```

Note that if (*current t*) = *None* then either (*left t*) or (*right t*) is empty.

3.2. Multi-tape Machines

We shall consider machines with three possible moves for the head: L (left) R (right) and N (None).

```
inductive move : Type :=| L : move | R : move | N : move.
```

The notion of Turing machine is parametric over a tape alphabet `sig` and the number `tapes_no` of *additional* working tapes (i.e. we assume the machine always has at least one tape). Formally, it is a record composed of a finite set of *states*, a transition function *trans*, a *start* state, and a set of halting states identified by a boolean function. To encode the alphabet and the states, we exploit the FinSet library of Matita, described in Section 2.

```
record mTM (sig:FinSet) (tapes_no:nat) : Type :=
{ states : FinSet;
  trans : states ×(Vector (option sig) (S tapes_no)) →
    states × (Vector ((option sig) ×move) (S tapes_no));
  start: states;
  halt : states →bool }.
```

The transition function takes in input a pair $\langle q, \vec{a} \rangle$ where q is the current internal state and $\vec{a}$ is a vector of optional symbols under the tape heads; it returns a pair $\langle q', \vec{p} \rangle$ where q' is a new internal state and $\vec{p}$ is a vector of *actions*. Each action is a pair $\langle b, m \rangle$ composed of a new *optional* character b and a move m. The new character is optional since we want to give the possibility to leave the tape untouched, that is particularly convenient when we are in an overflow position on some tape[4].

Executing an action $p = \langle b, m \rangle$ on a tape simply consists in writing b and moving the tape according to m:

```
definition tape_move_mono :=λsig,t,mv.
  tape_move sig (tape_write sig t (\fst mv)) (\snd mv).
```

Writing a symbol is straightforward:

```
definition tape_write :=λsig.λt: tape sig.λs:option sig.
  match s with [ None ⇒t | Some s0 ⇒midtape ? (left ? t) s0 (right ? t)].
```

The move operation is slightly more complex; we start by splitting it in simpler operations, according to the move

```
definition tape_move :=λsig.λt: tape sig.λm:move.
  match m with [ R ⇒tape_move_right ? t | L ⇒tape_move_left ? t | N ⇒t ].
```

Then, `tape_move_right` is defined in the following way (`tape_move_left` is analogous); note in particular that trying to move right when we are in an empty tape or in a right overflow situation has no effect.

[4]The type of actions has changed w.r.t.[8]; the point is that in mono-tape machines overflow situations are essentially pathological, while in multi-tape machines it is extremely convenient to exploit the border of the tapes as natural delimiters of data types stored on dedicated working tapes.

```
definition tape_move_right :=λsig:FinSet.λt:tape sig.
  match t with
  [ niltape ⇒ niltape sig
  | rightof _ _ ⇒ t
  | leftof a rs ⇒ midtape sig [ ] a rs
  | midtape ls a rs ⇒
    match rs with
    [ nil ⇒ rightof sig a ls
    | cons a0 rs0 ⇒ midtape sig (a::ls) a0 rs0
    ]
  ].
```

Finally, moving a vector of tapes w.r.t. a vector of actions simply consists in *mapping* the corresponding mono-tape operation:

```
definition tape_move_multi :=λsig,n,ts,actions.
  pmap_vec ...(tape_move_mono sig) n ts actions.
```

3.3. Configurations and computations

A *configuration* relative to a given set of `states` and an alphabet `sig` is a record composed of a current internal state `cstate` and a vector `ctapes` of tapes.

```
record mconfig (sig,states:FinSet) (n:nat): Type :=
{ cstate : states;
  ctapes : Vector (tape sig) (S n) }.
```

To perform a transition *step* between two configurations we must first of all extract from the input configuration `c` the current state and the vector of characters under the tape heads; then, we apply the transition function to get a new state `news` and an `action`, and finally build a new configuration composed by `news` and the tapes resulting by executing the given `action`. This is summarized by the following code:

```
definition step :=λsig.λn.λM:mTM sig n.λc:mconfig sig (states ?? M) n.
  let ⟨news,actions⟩ :=trans sig n M ⟨cstate ...c,current_chars ...(ctapes ...c)⟩ in
  mk_mconfig ...news (tape_move_multi sig ? (ctapes ...c) actions).
```

where `current_chars` returns the symbols currently under the tape heads:

```
definition current_chars :=λsig.λn.λtapes. vec_map ?? (current sig) (S n) tapes.
```

A computation is an iteration of the step function until a final internal state is met. In Matita, we may only define total functions, hence we provide an upper bound to the number of iterations, and return an optional configuration

depending on the fact that the halting condition has been reached or not. To this purpose, we use the following generic iterator[5] `loop` (inspired by [17]):

```
let rec loop (A:Type) n (f:A→A) p a on n :=
  match n with
  [ O ⇒ None ?
  | S m ⇒ if p a then (Some ? a) else loop A m f p (f a) ].
```

The transformation between configurations relative to Turing machine M is:

```
definition loopM :=λsig,n.λM:mTM sig n.λi,cin.
  loop ? i (step sig n M) (λc.halt sig n M (cstate ...c)) cin.
```

3.4. Semantics

The usual notion of computation for Turing machines is defined in relation with given input and output functions, providing the initial tape encoding and the final read-back function. As we know from Kleene's normal form, the output function is particularly important: the point is that our notion of Turing machine is monotonically increasing w.r.t. tape consumption, with the consequence that the transformation relation between configurations is decidable. However, input and output functions are extremely annoying when composing machines and we would like to get rid of them as far as possible.

Our solution is to define the semantics of a Turing machine by means of a relation between the input tape and the final tape (possibly embedding the input and output functions): in particular, we say that a machine M *realizes* a relation R between tapes ($M \models R$), if for all t_1 there exists a computation leading from $\langle q_0, t_1\rangle$, to $\langle q_f, t_2\rangle$ and $t_1\ R\ t_2$, where q_0 is the initial state, q_f is some halting state of M, and t_2 is the tape resulting from the computation.

```
definition initc :=λsig,n.λM:mTM sig n.λtapes.
  mk_mconfig sig (states sig n M) n (start sig n M) tapes.

definition Realize :=λsig,n.λM:mTM sig n.λR:relation (Vector (tape sig) ?).
∀t.∃i.∃outc.
  loopM sig n M i (initc sig n M t) = Some ? outc ∧ R t (ctapes ...outc).
```

It is natural to wonder why we use relations on tapes, and not on configurations. The point is that different machines may easily *share* tapes, but they can hardly share their internal states. Working with configurations would force us to an input/output recoding between different machines that is precisely what we meant to avoid.

The previous notion of realizability implies termination. It is useful to define a weaker notion (weak realizability, denoted $M \Vdash R$), asking that $t_1\ R\ t_2$ *provided* there is a computation between t_1 and t_2. It is easy to prove that

[5] For us, it was gratifying to discover that the small library of lemmas relative to the loop iterator that we develop for the mono-tape case, did not require any extension or modification after switching to the multi-tape case.

termination together with weak realizability imply realizability (we shall use the notation $M \downarrow t$ to express the fact that M terminates on input tapes t).

```
definition WRealize :=λsig,n.λM:mTM sig n.λR:relation (Vector (tape sig) ?).
∀t,i,outc.
  loopM sig n M i (initc sig n M t) = Some ? outc →R t (ctapes ??? outc).

definition Terminate :=λsig,n.λM:mTM sig n.λt. ∃i,outc.
  loopM sig n M i (initc sig n M t) = Some ? outc.

lemma WRealize_to_Realize : ∀sig,n .∀M: mTM sig n.∀R.
  (∀t.M ↓ t) →M ||= R →M |= R.
```

3.5. A canonical relation

For every machine M we may define a canonical relation, that is the smallest relation weakly realized by M

```
definition R_mTM :=λsig,n.λM:mTM sig n.λq.λt1,t2.
∃i,outc. loopM ? n M i (mk_mconfig ...q t1) = Some ? outc ∧ t2 = (ctapes ...outc).

lemma Wrealize_R_mTM : ∀sig,n.∀M:mTM sig n.
  M ||= R_mTM sig n M (start sig n M).

lemma R_mTM_to_R: ∀sig,n.∀M:mTM sig n.∀R. ∀t1,t2.
  M ||= R →R_mTM ?? M (start sig n M) t1 t2 →R t1 t2.
```

3.6. The Nop Machine

As a first, simple example, we define a Turing machine performing no operation (we shall also use it in the sequel to force, by sequential composition, the existence of a unique final state).

The machine has a single state that is both initial and final: we define the states as $initN\ 1$, that is the interval of natural numbers less than 1. This is actually a sigma type containing a natural number m and an (irrelevant) proof that m is smaller than n.

```
definition nop_states :=initN 1.
definition start_nop : initN 1 :=mk_Sig ?? 0 (le_n ...1).
```

The transition function is irrelevant, since it will never be executed: we define it as a map returning a vector of dummy actions.

```
definition null_action :=λsig.λn.
  Vector_of_list (make_list (option sig ×move) (⟨None ?,N⟩) (S n)).

definition nop :=
  λalpha:FinSet.λn.mk_mTM alpha n nop_states
    (λp.let ⟨q,a⟩ :=p in ⟨q,null_action sig n)⟩) start_nop (λ_.true).
```

The semantic relation *R_nop* characterizing the machine is just the identity; the proof that the machine realizes *R_nop* is entirely straightforward.

```
definition R_nop :=λalpha,n.λt1,t2:Vector (tape alpha) (S n).t2 = t1.

lemma sem_nop: ∀alpha,n.nop alpha n⊨ R_nop alpha n.
```

4. Composing Machines

Turing machines are usually reputed to suffer for a lack of compositionality. Our semantic approach, however, allows us to compose them in relatively easy ways. This will give us the opportunity to reason at a higher level of abstraction, rapidly forgetting their low level architecture.

4.1. Sequential composition

The sequential composition $M_1 \cdot M_2$ of two Turing machines M_1 and M_2 is a new machine having as states the disjoint union of the states of M_1 and M_2. The initial state is the (injection of the) initial state of M_1, and similarly the halting condition is inherited from M_2; the transition function is essentially the disjoint sum of the transition functions of M_1 and M_2, plus a transition leading from the final states of M_1 to the (old) initial state of M_2 (here it is useful to have the possibility of not moving the tape).

```
definition seq_trans :=λsig,n. λM1,M2 : mTM sig n.
λp. let ⟨s,a⟩ :=p in
  match s with
  [ inl s1 ⇒
      if halt sig n M1 s1 then ⟨inr ...(start sig n M2), null_action sig n⟩
      else let ⟨news1,m⟩ :=trans sig n M1 ⟨s1,a⟩ in ⟨inl ...news1,m⟩
  | inr s2 ⇒ let ⟨news2,m⟩ :=trans sig n M2 ⟨s2,a⟩ in ⟨inr ...news2,m⟩
  ].

definition seq :=λsig,n. λM1,M2 : mTM sig n.
  mk_mTM sig n
    ((states sig n M1) ⊕ (states sig n M2))
    (seq_trans sig n M1 M2)
    (inl ...(start sig n M1))
    (λs.match s with [ inl _ ⇒ false | inr s2 ⇒ halt sig n M2 s2]).
```

If $M_1 \models R_1$ and $M_2 \models R_2$ then

$$M_1 \cdot M_2 \models R_1 \circ R_2$$

that is a very elegant way to express the semantics of sequential composition. The proof of this fact, however, is not as straightforward as one could expect. The point is that M_1 works with its own internal states, and we should "lift" its computation to the states of the sequential machine. To have an idea of the kind of results we need, here is one of the key lemmas:

```
lemma loop_lift : ∀A,B,k,lift,f,g,h,hlift,c,c1.
  (∀x.hlift (lift x) = h x) →
  (∀x.h x = false →lift (f x) = g (lift x)) →
  loop A k f h c = Some ? c1 →
  loop B k g hlift (lift c) = Some ? (lift ...c1).
```

It says that the result of iterating a function g starting from a lifted configuration *lift* c is the same (up to lifting) as iterating a function f from c provided that

1. a base configuration is halting if and only if its lifted counterpart is halting;
2. f and g commute w.r.t. lifting on every non-halting configuration.

4.2. If then else

The next machine we define is an if-then-else composition of three machines M_1, M_2 and M_3 respectively implementing a boolean test, and the two conditional branches. We found convenient to store the result of the boolean test in an internal state of the first machine instead than on a tape: in particular we expect to end up in a distinguished final state *qacc* if the test is successful, and in a different state otherwise. This special state *qacc* must be explicitly mentioned when composing the machines. The definition of the if-then-else machine is then straightforward: the states of the new machine are the disjoint union of the states of the three composing machines; the initial state is the initial state of M_1; the final states are the final states of M_2 and M_3; the transition function is the union of the transition functions of the composing machines, where we add new transitions leading from *qacc* to the initial state of M_2 and from all other final states of M_1 to the initial state of M_2.

```
definition if_trans :=λsig,n. λM1,M2,M3 : mTM sig n. λq:states sig n M1.
λp. let ⟨s,a⟩ :=p in
  match s with
  [ inl s1 ⇒
     if halt sig n M1 s1 then
       if s1==q then ⟨inr ...(inl ...(start sig n M2)), null_action ??⟩
       else ⟨inr ...(inr ...(start sig n M3)), null_action ??⟩
     else let ⟨news1,m⟩ :=trans sig n M1 ⟨s1,a⟩ in ⟨inl ...news1,m⟩
  | inr s' ⇒
     match s' with
     [ inl s2 ⇒ let ⟨news2,m⟩ :=trans sig n M2 ⟨s2,a⟩ in ⟨inr ...(inl ...news2),m⟩
     | inr s3 ⇒ let ⟨news3,m⟩ :=trans sig n M3 ⟨s3,a⟩ in ⟨inr ...(inr ...news3),m⟩
     ]].
```

```
definition ifTM :=λsig,n. λcondM,thenM,elseM : mTM sig n.
  λqacc: states sig n condM.
  mk_mTM sig n
    ((states sig n condM) ⊕ (states sig n thenM) ⊕ (states sig n elseM))
    (if_trans sig n condM thenM elseM qacc)
    (inl ...(start sig n condM))
    (λs.match s with
      [ inl _ ⇒ false
      | inr s' ⇒ match s' with
        [ inl s2 ⇒ halt sig n thenM s2
        | inr s3 ⇒ halt sig n elseM s3 ]]).
```

Our realizability semantics is defined on tapes, and not configurations. In order to observe the accepting state we need to define a suitable variant that we call *conditional realizability*, denoted by $M \models [q : R_1, R_2]$. The idea is that M realizes R_1 if it terminates the computation on q, and R_2 otherwise.

```
definition accRealize :=λsig,n.λM:mTM sig n.λacc:states sig n M.λRtrue,Rfalse.
∀t.∃i.∃outc.
  loopM sig n M i (initc sig n M t) = Some ? outc ∧
    (cstate ??? outc = acc →Rtrue t (ctapes ??? outc)) ∧
    (cstate ??? outc ≠acc →Rfalse t (ctapes ??? outc)).
```

The semantics of the if-then-else machine can be now elegantly expressed in the following way:

```
lemma sem_if: ∀sig,n.∀M1,M2,M3:mTM sig n.∀Rtrue,Rfalse,R2,R3,acc.
  M1 ⊨ [acc: Rtrue,Rfalse] →M2 ⊨R2 →M3 ⊨R3 →
    ifTM sig n M1 M2 M3 acc ⊨ (Rtrue ∘R2) ∪(Rfalse ∘R3).
```

It is also possible to state the semantics in a slightly stronger form: in fact, we know that if the test is successful we shall end up in a final state of M_2 and otherwise in a final state of M_3. If M_2 has a single final state, we may express the semantics by a conditional realizability over this state. A simple way to force a machine to have a unique final state is to sequentially compose it with the nop machine. Then, it is possible to prove the following result (the conditional state is a suitable injection of the unique state of the nop machine):

```
lemma acc_sem_if: ∀sig,n,M1,M2,M3,Rtrue,Rfalse,R2,R3,acc.
  M1 ⊨ [acc: Rtrue, Rfalse] →M2 ⊨R2 →M3 ⊨R3 →
  ifTM sig n M1 (single_finalTM ...M2) M3 acc ⊨
     [inr ...(inl ...(inr ...start_nop)): Rtrue ∘R2, Rfalse ∘R3].
```

4.3. While

The last machine we are interested in, implements a while-loop over a body machine M. Its definition is really simple, since we have just to add to M a single transition leading from a distinguished final state q back to the initial state.

```
definition while_trans :=λsig,n. λM: mTM sig n. λq:states sig n M. λp.
  let ⟨s,a⟩ :=p in
  if s == q then ⟨start ?? M, null_action ??⟩
  else trans ?? M p.

definition whileTM :=λsig,n. λM: mTM sig n. λqacc: states ?? M.
  mk_mTM sig n
    (states ?? M)
    (while_trans sig n M qacc)
    (start sig n M)
    (λs.halt sig n M s ∧ ¬ s==qacc).
```

More interesting is the way we can express the semantics of the while machine: provided that $M \models [q : R_1, R_2]$, the while machine (relative to q) weakly realizes $R_1^* \circ R_2$:

```
theorem sem_while: ∀sig,n,M,acc,Rtrue,Rfalse.
 halt sig n M acc = true →
 M ⊨ [acc: Rtrue,Rfalse] →
   whileTM sig n M acc |⊨ (star ? Rtrue) ∘Rfalse.
```

In this case, the use of weak realizability is essential, since we are not guaranteed to exit the while loop, and the computation can actually diverge. Interestingly, we can reduce the termination of the while machine to the well foundedness of $Rtrue$:

```
theorem terminate_while: ∀sig,n,M,acc,Rtrue,Rfalse,t.
 halt sig n M acc = true →
 M ⊨ [acc: Rtrue,Rfalse] →
 WF ? (inv ...Rtrue) t → whileTM sig n M acc ↓t.
```

5. Basic Machines

An important lesson learned in [8] was to avoid modelling relatively complex operations by directly writing a corresponding Turing machine. While writing the code is usually not very complex, proving its correctness can easily become a nightmare, due to the complexity of specifying and reasoning about internal states of the machines and all intermediate configurations. A much better approach consists in specifying a small set of basic machines, and define all other machines by means of the compositional constructs of the previous section. In this way, we may immediately forget about Turing machines' internals, since the behavior of the whole program only depends on the behavior of its components.

We can divide the basic machines in two sets: machines acting on a single tape, and machines acting in parallel over two tapes.

The set of machines acting on a single table is essentially a subset of the machines[6] already considered in [8]:

write c write the character c on the tape at the current head position

move D move the head one step towards direction D

test_char f perform a boolean test f on the current character ending in state `tc_true` or `tc_false` according to the result of the test

[6]Most of them are actually *families* of machines, indexed over suitable input arguments.

The specification of these machines is straightforward. We typically generalized them to the multi-tape case by means of a generic "injection" operation allowing to operate on a specified tape.

Typical basic machines working in parallel on two tapes are the following:

par_move_step i j D make a parallel D move on tapes i and j

compare_step i j compare head-characters on tapes i and j, ending up in state `comp1` if they are equal, and `comp2` otherwise.

copy_char_N i j copy head-character on tape i to tape j without advancing the tapes.

It is interesting to observe that the swap operation, that played an important role when working on mono-tape machines, is absolutely useless in the multi-tape case.

5.1. Composing machines

Let us see an example of how we can use the previous bricks to build more complex functions. In Turing machines, moving the head to a specific position on the tape is a very frequent operation. In particular, our universal machine needs to return to one end of a given tape on several different occasions. We are now going to show how to write a machine `move_to_end` that moves the head towards a given direction until the end of the tape is met.

A step of the machine essentially consists of a move operation, but guarded by a conditional test; then we shall simply wrap a while machine around this step.

```
definition mte_step :=λalpha,D.
ifTM ? (test_null alpha) (single_finalTM ? (move alpha D)) (nop ?) tc_true.
```

If the test succeeds we expect to end up in the single final state of the "then" machine, that is the following `mte_acc` state:

```
definition mte_acc : ∀alpha,D.states ? (mte_step alpha D) :=
λalpha,D.(inr ...(inl ...(inr ...start_nop))).
```

In this case, the input tape t_1 must be a midtape and we expect to end up with a tape t_2 resulting from t_1 by performing the given move. This is expressed by the following relation:

```
definition R_mte_step_true :=λalpha,D,t1,t2.
  ∃ls,c,rs. t1 = midtape alpha ls c rs ∧ t2 = tape_move ? t1 D.
```

Conversely, if the test fails it means that we are in overflow, and we do nothing:

```
definition R_mte_step_false :=λalpha.λt1,t2:tape alpha.
  current ? t1 = None ? ∧ t1 = t2.
```

The semantics of the `mte_step` machine is hence expressed as follows:

```
lemma sem_mte_step :
  ∀alpha,D.mte_step alpha D ⊨
   [ mte_acc ...: R_mte_step_true alpha D, R_mte_step_false alpha ] .
```

Here is the full `move_to_end` program:

```
definition move_to_end :=λsig,D.whileTM sig (mte_step sig D) (mte_acc ...).

definition R_move_to_end_l :=λsig,int,outt.
  (current ? int = None ? →outt = int) ∧
  ∀ls,c,rs.int = midtape sig ls c rs →
  outt = mk_tape ? [ ] (None ?) (reverse ? ls@c::rs).

definition R_move_to_end_r :=λsig,int,outt.
  (current ? int = None ? →outt = int) ∧
  ∀ls,c,rs.int = midtape sig ls c rs →
  outt = mk_tape ? (reverse ? rs@c::ls) (None ?) [ ].

lemma sem_move_to_end_l : ∀sig. move_to_end sig L ⊨ R_move_to_end_l sig.

lemma sem_move_to_end_r : ∀sig. move_to_end sig R ⊨ R_move_to_end_r sig.
```

We do not give semantics for the useless non-terminating machine obtained by instantiating `move_to_end` to the N direction.

6. Normal Turing machines

A normal Turing machine is a mono-tape machine where:

1. the tape alphabet is $\{0, 1\}$;
2. the finite states are supposed to be an initial interval of the natural numbers.

By convention, we assume the starting state is 0.

```
record normalTM : Type :=
{ no_states : nat;
  pos_no_states : (0 < no_states);
  ntrans : (initN no_states)×Option bool →(initN no_states)×(Option bool)×Move;
  nhalt : initN no_states →bool}.
```

We may easily define a coercion transforming a normal TM into a traditional Machine.

A *normal configuration* `nconfig n` is a configuration for a normal machine with n states, that is a record composed of an element in `initN n` (representing the state) and a single boolean tape.

```
definition nconfig :=λn. config FinBool (initN n).
```

6.1. Tuples

By the results on FinSets of Section 2 we know that every function f between two finite sets A and B can be described by means of a finite graph of pairs $\langle a, f\ a\rangle$. Hence, the transition function of a normal Turing machine can be described by a finite set of tuples $\langle\langle i, c\rangle, \langle j, d, m\rangle\rangle$ of the following type:

$$(\texttt{initN}\ n \times \mathit{option\ bool}) \times (\texttt{initN}\ n \times (\mathit{option\ bool}) \times \mathit{move})$$

Unfortunately, this description is not directly suitable for a Universal Machine, since such a machine must work with a fixed set of states, while the size on n is unknown. Hence, we must pass from natural numbers to a representation for them on a finitary, e.g. binary, alphabet. In general, we shall associate to a pair $\langle\langle i, c\rangle, \langle j, d, m\rangle\rangle$ a tuple with the following syntactical structure

$$|\ w_i\ x\ w_j\ y\ z$$

where

1. "|" is a special character used as a separator;
2. w_i and w_j are lists of booleans representing the states i and j;
3. x represents the symbol c: it is a special character `null` if $c =$ None and is the boolean b if $c =$ Some b; the character y represents d in the same way;
4. similarly, for moves we map N into `null`, and L and R on the two booleans false and true.

This encoding of tuples is much simpler than the one described in [8]. In fact, working with a multi-tape machine, the set of tuples will be stored on a dedicated tape, that allows to spare a few delimiters; moreover, having multiple heads, we may avoid the annoying use of *markers* to remember tape positions during comparison and copying operations on strings (performed over different tapes).

```
definition mk_tuple :=λqin,cin,qout,cout,mv.
  bar :: qin @ cin :: qout @ [cout; mv].
```

We codified a state i in a machine with n states as a list of n bits where the bit at position i is set to 1 and the other bits are all 0. We also add an initial bit set to 1 if the state is final and to 0 otherwise. For instance, if you have four states and states 2 and 4 are final, they would be encoded as follows:

$$1 = 01000, 2 = 10100, 3 = 00010, 4 = 10001$$

The function computing such a representation is called `bits_of_state` (we omit the obvious definition). As a matter of fact, the actual encoding of states is not very important, but it is convenient to assume that (a) all states (and hence all tuples for a given machine) have a fixed, uniform length, and (b) it is easy to extract from the state the fact that it is final or not.

In the same way, the two functions `low_char` and `low_mv` compute the low level representation of characters and moves of the normal machine.

```
definition low_char :=λc.
  match c with [ None ⇒ null | Some b ⇒ bit b ].

definition low_mv :=λm.
  match m with [ R ⇒ bit true | L ⇒ bit false | N ⇒ null ].
```

It is simple to prove that such encoding functions (as well as `bits_of_state`) are injective (and hence invertible).

Combining all these functions together, it is easy to write a function `tuple_encoding` mapping a machine tuple into a list of characters. More precisely, the function `tuple_encoding` takes as arguments a number n of tapes, a halting test function `h:Nat_to n→bool` relative to some normal machine M, a tuple t for the same machine and returns the encoding for t.

6.2. Input configurations

An input configuration is a string corresponding to some input $\langle q, a\rangle$ for the transition function of the normal machine, preceded by the symbol "|" (`bar`).

If `bar::qin@cin::qout@[cout;mv]` is a tuple, then `bar::qin@[cin]` is an input configuration, and similarly `bar::qout@[cout]` is the (next) input configuration.

In the universal machine, input configurations will be stored on a dedicated tape. The following predicate captures the relevant syntactical characteristics of input configurations:

```
definition is_config : :=λn,cfg.∃qin,cin.
 only_bits qin ∧ cin ≠bar ∧ |qin| = S n ∧ cfg = bar::qin@[cin].
```

An important property of tuples is that, if they derive from a transition function, they are deterministic: for any input configuration there is only one corresponding output configuration.

```
lemma deterministic: ∀M:normalTM.∀l,t1,t2,c,out1,out2.
  l = graph_enum ?? (ntrans M) →
  mem ? t1 l →mem ? t2 l →
  is_config (no_states M) c →
  tuple_encoding ? (nhalt M) t1 = (c@out1) →
  tuple_encoding ? (nhalt M) t2 = (c@out2) →
  out1 = out2.
```

6.3. The table of tuples

The list of all tuples, concatenated together, provides the low level description of the normal Turing machine to be interpreted by the Universal Machine: we call it a *table*.

```
definition table_TM :=λn,l,h.flatten ? (tuples_list n h l).
```

The main lemma relating a table to the list of its tuples `l` is the following one, stating that for any input configuration `c`, if `c` is matched inside the table, then it is *followed* by a string `out` such that `c@out` is the encoding of some tuple `t` in `l`.

```
lemma table_to_list: ∀n,l,h,c. is_config n c →
  ∀ll,lr.table_TM n l h = ll@c@lr →
    ∃out,lr0,t. lr = out@lr0 ∧ tuple_encoding n h t = (c@out) ∧ mem ? t l.
```

7. The Universal Machine

In this section we define a multi-tape Universal Machine (UM) for normal Turing machines. Let M be the normal machine to be simulated; the universal machine will work with three tapes, with the following functions:

- the object tape (obj) is used to simulate the tape of M;
- the configuration (cfg) tape contains the encoding of the current state of M, concatenated with a copy of the head symbol (that is, at each step of M, the input for its transition function)
- the program (prg) tape contains the table of tuples relative to M.

Each step of the normal machine M is simulated by the following sequence of operations:

```
definition unistep :=
  match_m cfg prg UniS 2 ·  (* find a matching tuple *)
  restart_tape cfg ·        (* move to the beginning of the cfg tape *)
  mmove cfg ? 2 R ·         (* skip initial bar on cfg *)
  copy prg cfg UniS 2 ·     (* copy the tuple output to cfg *)
  exec_move                 (* update the obj tape *)
```

where `mmove` is the (non-primitive) multi-tape version of `move`. The behaviour of the other functions is summarized in Figure 1, where we also also emphasize the expected positions of the tape heads after each operation (that is a quite delicate semantical issue). An execution of `unistep` is shown in Figure 2.

The body of the universal machine `uni_body` consists of checking the termination condition (i.e. examining the first bit of the current state, to see if it is a final state) and, if the condition is not met, executing `uni_step`.

The universal machine is simply a while over `uni_body`:

```
definition uni_body :=
  ifTM ?? (mtestR ? cfg 2 stop)
    (single_finalTM ?? unistep)
    (nop ...) (mtestR_acc ? cfg 2 stop).

definition universalTM :=whileTM ? uni_body us_acc.
```

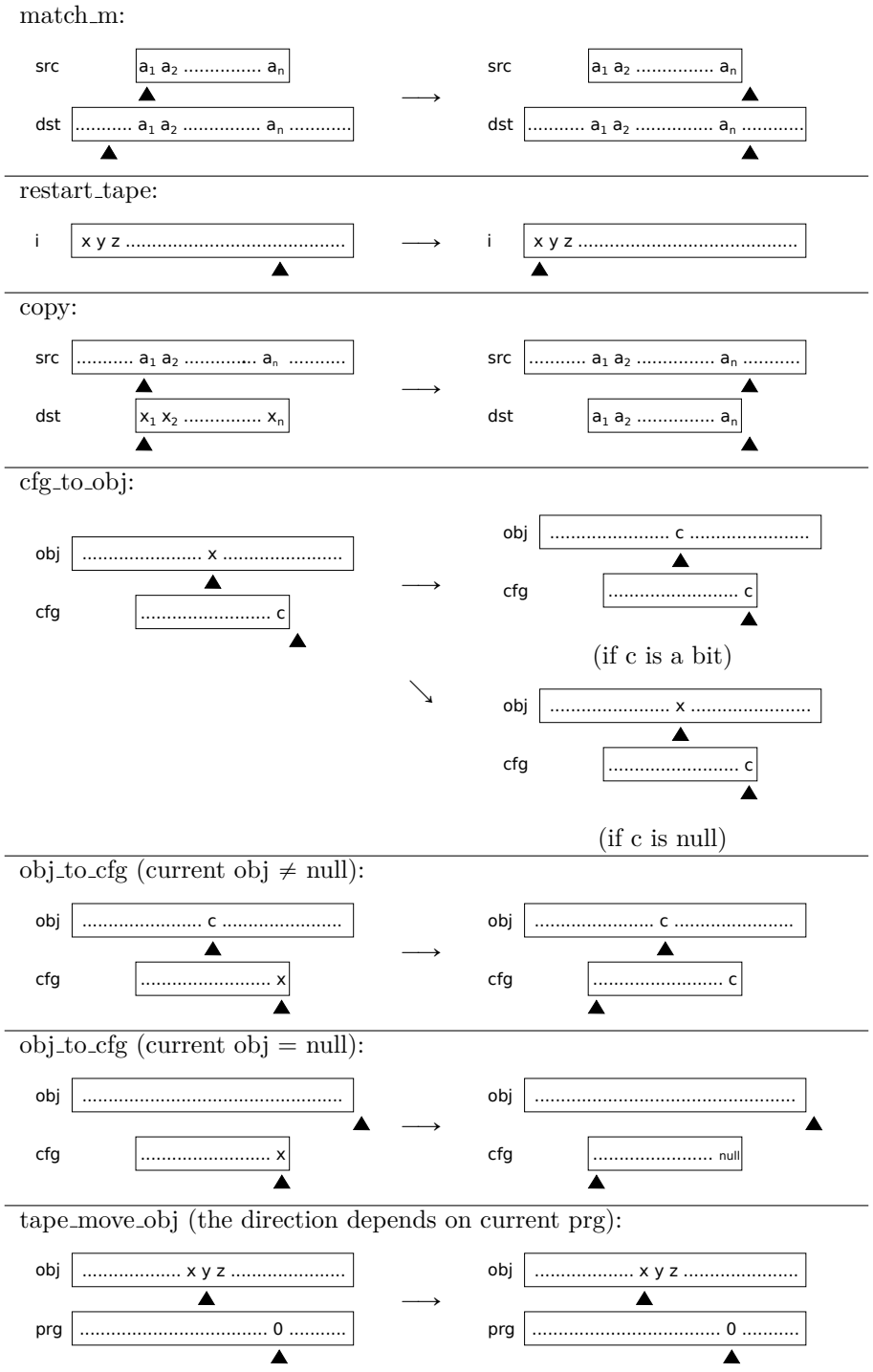

Figure 1: Machines used in `unistep`.

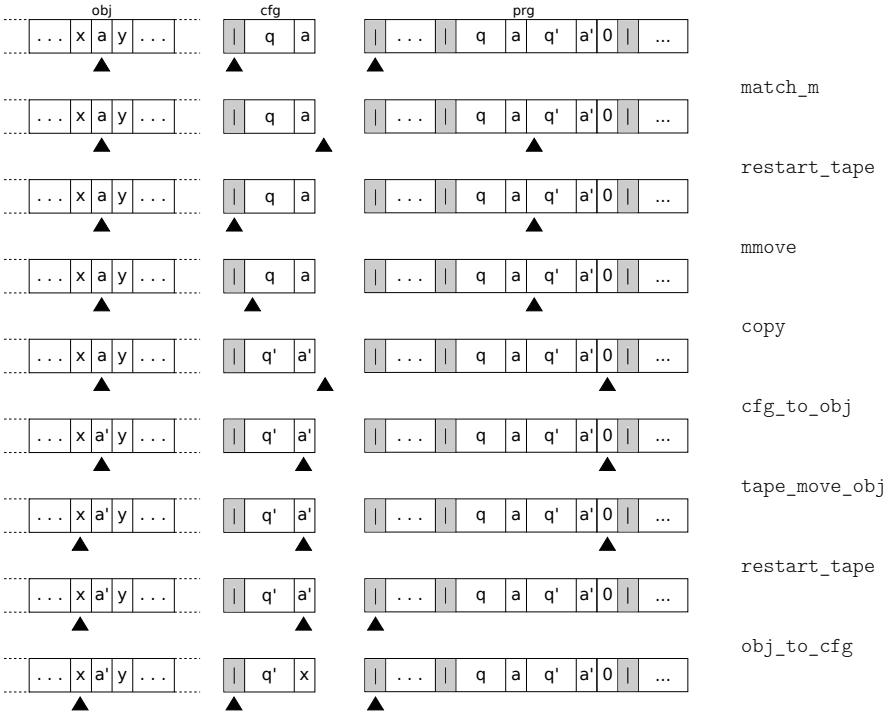

Figure 2: Sample execution of `unistep`.

8. Matching

The matching operation involves repeatedly comparing the current configuration to the input configuration of each tuple on the prg tape: when the two are different, we will move to the next tuple and loop; otherwise we will stop, keeping the head of the prg tape at the beginning of the output configuration of the matching tuple.

Matching is the most complex operation used in the universal machine, since it is obtained by iterating, in a while loop, the string comparison machine, which is made of a while loop itself. In our mono-tape formalization, its implementation was further complicated by the need to consider various kinds of delimiters, and to set marks needed to move back and forth between the two strings under consideration.

8.1. String comparison

In a multi-tape setting we can compare two characters in a single step, without any need to store the first character in the internal state (let alone use the more complicated comparison technique adopted in [8] for technical reasons). A further difference, accounting for more simplification, is that the

new machine does not use delimiters to decide whether the comparison has been completed successfully, but terminates when one of the two tapes is over. This machine is obtained by means of a simple while iterating the `compare_step` basic machine.

```
definition compare :=λi,j,sig,n.
  whileTM ...(compare_step i j sig n) comp1.
```

More precisely, the compare machine running on tapes `i` and `j` will succeed if one of the two is a prefix of the other one. Since we plan to use the machine to compare an initial configuration in `prg` with the content of `cfg`, success corresponds to the case where the portion of those two tapes standing on the right of the head is found to be equal, until the end of tape `cfg` is reached.

Failure happens when at some point, the `k`-th characters on the right of the two tapes considered are different, in which case the heads are left on the two differing characters. These two outcomes are described by the semantics of `compare`:

```
definition R_compare :=
  λi,j,sig,n.λint,outt: Vector (tape sig) (S n).
  ...
  (∀ls,x,rs,ls0,rs0.
    nth i ? int (niltape ?) = midtape sig ls x rs →
    nth j ? int (niltape ?) = midtape sig ls0 x rs0 →
    (∃rs'.rs = rs0@rs' ∧
     outt = change_vec ??
            (change_vec ?? int
              (mk_tape sig (reverse sig rs0@x::ls) (option_hd sig rs')
                (tail ? rs')) i)
            (mk_tape sig (reverse sig rs0@x::ls0) (None ?) [ ]) j) ∨
    ...
    (∃xs,ci,cj,rs',rs0'.ci ≠cj ∧rs = xs@ci::rs' ∧rs0 = xs@cj::rs0' ∧
     outt = change_vec ??
            (change_vec ?? int (midtape sig (reverse ? xs@x::ls) ci rs') i)
            (midtape sig (reverse ? xs@x::ls0) cj rs0') j)).

lemma wsem_compare : ∀i,j,sig,n.i ≠j →i < S n →j < S n →
  compare i j sig n ⊨ R_compare i j sig n.
```

The semantics of `compare` as we formalized it also includes two more possible outcomes (here denoted by ellipses): a symmetric success case (due to the fact that the machine operates in the same way on the two tapes) and a base case stating the outcome of the machine when the termination condition is immediately verified and no iterations of `compare_step` are executed.

8.2. `match_step`

The `match_step` starts by performing a compare over the tapes cfg and prg, then it checks whether the halting condition is met: we want to stop if the head of cfg or prg has stopped over an uninitialized portion of the tape. In the former case, the compare operation was successful; in the latter case, we have finished examining a table without finding a matching tuple. Checking whether

the table has ended may seem useless, since it is supposed to represent a total transition function, but it will allow us to prove the termination of match_step for all inputs (including ill-formed ones).

If the termination condition is not met, in order to prepare for a new iteration of match_step we will move back to the start of tape cfg, and to the next tuple on tape prg. Since both the configuration on cfg and all the tuples in the tape start with a "|" character, we obtain the same result with a simpler machine by moving to the next character in prg, instead of moving to the next tuple: subsequent iterations will fail immediately until the next tuple, starting with "|", is found.

```
definition match_step :=λsrc,dst,sig,n.
  compare src dst sig n ·
     (ifTM ?? (partest sig n (match_test src dst sig ?))
      (single_finalTM ??
        (rewind src dst sig n · mmove dst ?? R))
      (nop ...)
      partest1).
```

Notice that the termination condition match_test will check whether either the source or the destination tape have ended.

The match_m machine is obtained by simply iterating match_step until the termination condition is met.

```
definition match_m :=λsrc,dst,sig,n.
  whileTM ...(match_step src dst sig n)
    (inr ?? (inr ?? (inl ...(inr ?? start_nop)))).
```

After a successful execution of match_m, the prg head will be at the beginning of the output of the matching tuple, and the cfg head will be at the right end of its tape in overflow position.

9. State update

We decompose the execution of an action in two simpler operations, which can be executed sequentially: in the first half, we will copy the output portion of the tuple selected by the match_m machine into the cfg tape, which will then contain the new current state and the new character to be written to the tape; the second half is concerned with updating the obj tape and obtaining the next input configuration.

The update of the state is performed, after returning the head on the cfg tape to the first character on the left (restart_tape), by the copy machine.

```
definition copy :=λsrc,dst,sig,n.
  whileTM ...(copy_step src dst sig n) copy1.
```

Similarly to the compare operation, copying is done one character at a time and stops when the end of the tape is reached on either the source or the destination. This is obtained by iterating the conditional machine copy_step in a while cycle. In our case, it is the destination (cfg) that is going to end first:

thus at the end of the execution, the cfg head will be at the right end of its tape in overflow position, and the prg head will have moved to the last character of the selected tuple (representing the direction towards which the obj head should move).

10. Tape update

Finally, the exec_move updates the tape of the simulated machine.

```
definition exec_move :=
  cfg_to_obj · tape_move_obj · restart_tape prg 2 ·obj_to_cfg.
```

The cfg_to_obj machine reads the new character to be written from the cfg tape (where it has just been written by copy) and writes it to the obj tape at the current position. However, if the character is not a bit, but the special value null, the prg tape will be left untouched.

tape_move_obj moves the obj head to the left (if the current character under the prg head is a 0 bit), to the right (if that character is a 1 bit), or does not move it at all (if that character is a null).

The restart_tape has the usual semantics, and is used in this case to return the prg head to its initial position, where it is expected to be at the next iteration of unistep.

The last machine obj_to_cfg updates cfg putting the new character read from obj, obtaining the next input configuration. If the obj head is in an overflow position, the special character null will be written to cfg instead. After this operation, the cfg head is also returned to its initial position.

The previous machines are relatively trivial; the interesting point, in this case, is to relate their low-level operational behaviour to the intended meaning, that is to mimic the execution of a move of the simulated machine.

11. Main Results

Given a configuration c for a normal machine M, the following function builds the corresponding "low level" representation, i.e. the three tapes [obj;cfg;prg] manipulated by the Universal Machine:

```
definition low_tapes: ∀M:normalTM.∀c:nconfig (no_states M).Vector ? 3 :=
λM:normalTM.λc:nconfig (no_states M).Vector_of_list ?
  [tape_map ?? bit (ctape ?? c)
  ;midtape ? [ ] bar
    ((bits_of_state ? (nhalt M) (cstate ?? c))@[low_char (current ? (ctape ?? c))])
  ;midtape ? [ ] bar (tail ? (table_TM ? (graph_enum ?? (ntrans M)) (nhalt M)))
  ].
```

The first tape is simply an embedding of the tape ctape ?? c in the alphabet of the Universal Machine: every bit b is transformed in the character bit b; the tape_map function is the obvious extension of the map functions to tapes:

```
definition tape_map :=λA,B:FinSet.λf:A→B.λt.
  mk_tape B (map ?? f (left ? t))
    (option_map ?? f (current ? t)) (map ?? f (right ? t)).
```

The second tape contains the current input configuration for the transition function discussed in Section 6.2.

Finally, the third tape is just the table of tuples for the normal machine `M`.

The semantics of the `unistep` machine can be expressed very elegantly in terms of `low_tapes`:

```
definition R_unistep_high :=λM:normalTM.λt1,t2.
∀c:nconfig (no_states M).
  t1 = low_tapes M c →t2 = low_tapes M (step ? M c).
```

Every relation over tapes can be reflected into a corresponding relation on their low-level representations:

```
definition low_R :=λM,q,R.λt1,t2:Vector (tape FSUnialpha) 3.
    ∀Mt. t1 = low_tapes M (mk_config ?? q Mt) →
    ∃cout. R Mt (ctape ...cout) ∧
    halt ? M (cstate ...cout) = true ∧t2 = low_tapes M cout.
```

Starting from the above semantics of the `unistep` machine, it is then easy (20 lines) to prove that, for any normal machine M, the Universal Machine weakly realizes the low level version of the canonical relation for M

```
theorem sem_universal: ∀M:normalTM. ∀q.
  universalTM |⊨ low_R M q (R_TM FinBool M q).
```

As a corollary of the latter result we obtain that, for any relation weakly realized by M, the universal machine weakly realizes its low level counterpart.

```
  M |⊨ R →universalTM |⊨ (low_R M (start ? M) R).
```

Termination is stated by the following result, whose proof (once we may rely the totality of the `unistep` function, is entirely straightforward (ten lines of code):

```
theorem terminate_UTM: ∀M:normalTM.∀t.
  M ↓ t →universalTM ↓(low_tapes M (mk_config ?? (start ? M) t)).
```

12. Discussion

We provided in this paper a formalization of the basic Theory of Turing machines, up to the definition of a universal machine and the proof of its correctness.

The work is a deep revisitation of [8]; in particular, we generalized the investigation from mono-tape machines to the multi-tape case. In spite of the fact that the meta-theory of multi-tapes machines is technically more complex, its formalization, taking advantage of the vector library of Matita, takes about the same space of the mono-tape case.

On the other side, the formalization of the universal machine is *sensibly* simplified. In Table 3 we compare the size, in lines, of the main files in the case of the mono-tape version described in [8] with respect to the multi-tape version described in this article. Note that the complex management of marks for copying/comparing strings in the mono-tape case has no counterpart in the multi-tape case. We should also stress that *termination* was still under development in [8], while it has been completely proved in the current version.

file name	mono	multi	content
`marks.ma`	901	0	management of marks
`alphabet.ma`	110	59	alphabet of the universal machine
`normalTM.ma`	319	257	normal Turing machines
`tuples.ma`	276	229	the table of tuples
`match.ma`	727	729	matching a tuple in the table
`uni_step_aux.ma`	778	521	auxiliary machines
`uni_step.ma`	585	458	emulation of a high-level step
`universal.ma`	394	120	the universal machine
total	4090	2373	

Figure 3: Main files and their dimension in lines

.

The universal machine presented in this article is a multi-tape machine simulating mono-tape *normal* machines. To provide a truly universal machine two auxiliary mappings are needed, one transforming a multi-tape machine to a mono-tape one (possibly enlarging the alphabet) and one mapping a mono-tape machine to a Normal machine. We implemented both these transformations and proved them correct; the proofs are not trivial: the first one takes about 2 KLOC, and the second one 1 KLOC. Both transformations are quite interesting, for different reasons, but their description would add very little to this article and, also due to space limitations, we prefer to postpone their discussion for a future work.

13. Related works

As we mentioned in the introduction, at the time we first submitted this article there was no other mechanizations of Turing Machines to compare with. Very recently, however, Xu, Zhang and Urban published an alternative formalization of the theory of Turing machines [35], partially inspired by our previous work [8].

The approach followed by Xu, Zhang and Urban is sensibly different from ours, and mostly based on Boolos et al. textbook [16]. Starting from a simple notion of Turing machine similar to our normal Turing machine, they consider a small hierarchy of higher level computation models, implementing each layer on the previous one. The intermediate layer is provided by abacus machines, that

are random access machines with an infinite number of registers, each able to store an arbitrarily large natural number. The highest level is the language of recursive functions with primitive recursion and minimization. The interpreter for Turing machines is written in this language, and then compiled into a Turing machine to give a truly universal machine.

At the semantic level, Xu, Zhang and Urban develop a Hoare-style technique to reason about concrete Turing machines (and abacus programs). A Hoare-triple $\{P\}\, M\, \{Q\}$ expresses the idea that a program M starting the computation with a tape satisfying P will terminate with a tape satisfying Q. As admitted by the authors themselves, the idea is very similar to our notion of realizability. As a matter of fact, we considered the possibility of expressing semantics in the form of triples. However, one is often interested in expressing the properties of the output tape *in terms* of the properties of the input tape, that amounts to saying that Q should be a relation, in which case there is no need for P at all (unless you are interested in expressing guards, an approach that we developed as well and finally did not use). From a practical point of view, our approach to the construction of Turing machines seems to be much more modular and compositional than the one in [35]. All our basic machines have very few states (up to 4) and all other machines are derived by composition: on the contrary, Xu, Zhang and Urban define very complex and monolithic machines with a lot of states (e.g. the copy machine has 15 states, and other machines with even more states are mentioned). This is admittedly due to a deficiency of their semantical approach[7].

Even if one is not interested - as we are, in the long term - to use Turing machines as a foundational model for Complexity Theory, the efficiency of the Universal machine remains an interesting issue. The machine described in this paper is fairly efficient: the slow-down with respect to the simulated program M is constant, where the constant depends polynomially on the size of M (we have not proved it formally yet, though). With some care, the transformations from multi-tape to mono-tape machines, and from mono-tape to normal machines should not change this complexity (one needs to exploit the fact that the size of all tapes but one is fixed along the computation). So, our interpreter *seems* to be *fair*, in the sense of [4] (that remains to be formally checked).

The complexity of the universal machine in [35] is much worse. The simulation of abacus machines on Turing Machines results in a cubic slow-down: fetching the parameters of an instruction has a complexity that is proportional to the dimension of the tape, and such a dimension grows quadratically with time. The cost of simulating recursive functions on abacus machines is not so clear, and the issue is not discussed in [35]; in particular, a pretty delicate issue is related to the adoption of a unary notation for the natural numbers, that could sensibly alter the cost model for recursive functions.

From the point of view of the complexity of the formalization, our approach seems to be sensibly more compact than the one in [35]. Even counting the

[7] see the discussion at page 158

correctness proofs for the operations transforming a multi-tape machine into a normal machine our development takes, in the whole, less than 6,000 lines (plus about 4,700 lines for the formalization of mono and multi-tape machines).

According to the figures provided by the authors in [35], the formalization of the universal machine takes by itself more than 10,000 lines of code, plus 4,600 lines for the formalization of abacus machines and their translation to Turing machines, and 2,800 lines for the formalization of recursive functions and their implementation on abacus machines.

14. Conclusions

Proving the correctness of a universal Turing machine is a quite complex task: Norrish describes it as a "daunting prospect" ([29], page 310), and Xu, Zhang and Urban admit that "is not a project one wants to undertake too many times" ([35], page 160). Still, it is an interesting exercise, stressing the potentialities of interactive provers on a relatively elementary topic, and we hope more people belonging to different proof assistant communities will undertake the job.

For us, the long term goal is to define a solid ground that could allow us to talk about Complexity at a higher level of abstraction. Due to their finitistic nature, the notions of time and space complexity for Turing machines are particularly clear and universally accepted, that is not the case for other, higher level programming paradigms. So, even modern textbooks in Complexity Theory (see e.g. [3]) start with introducing Turing Machines, possibly to claim, immediately after, that the computational model *does not matter*. The natural question is to understand *what* matters, then.

The final goal of our research would be to obtain a formal, axiomatic treatment of Complexity Theory at a *comfortable* level of abstraction, expressing it in terms of a few basic assumptions (e.g. the ability to perform bound interpretation with a given, possibly parametric complexity) and suitable closure properties. We plan to derive such an abstract framework from a *reverse* investigation of major results of Complexity Theory, reconstructing from proofs the key assumptions underlying them (see [5] for a first, preliminary result in this direction).

Eventually, the resulting axiomatization will need to be validated with respect to concrete cost models, and in particular with respect to Turing machines, that provides the actual foundation for Complexity Theory. Hence, in conjunction with the "reverse" approach, it is also important to promote a more traditional forward approach, deriving out of concrete models the key notions and results for the formal study of their complexity aspects. The work in this paper, is meant to be a contribution along this second line of research.

References

[1] R. Amadio, A. Asperti, N. Ayache, B. Campbell, D. Mulligan, R. Pollack, Y-Régis-Gianas, C-Sacerdoti Coen, and I. Stark. Certified Complexity. *Procedia CS*, 7:175–177, 2011.

[2] Andrew W. Appel, Robert Dockins, and Xavier Leroy. A List-Machine Benchmark for Mechanized Metatheory. *J. Autom. Reasoning*, 49(3):453–491, 2012.

[3] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge Univ. Press, 2009.

[4] Andrea Asperti. The intensional content of Rice's theorem. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), January 7-12, 2008, San Francisco, California, USA*, pages 113–119. ACM, 2008.

[5] Andrea Asperti. A formal proof of borodin-trakhtenbrot's gap theorem. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2013.

[6] Andrea Asperti and Claudio Sacerdoti Coen. Some considerations on the usability of interactive provers. In *Intelligent Computer Mathematics, 10th International Conference, Paris, France, July 5-10, 2010*, volume 6167 of *Lecture Notes in Computer Science*, pages 147–156. Springer, 2010.

[7] Andrea Asperti and Jeremy Avigad (eds). Special Issue on Interactive Theorem Proving and the Formalisation of Mathematics. *Mathematical Structures in Computer Science*, 21(4), 2011.

[8] Andrea Asperti and Wilmer Ricciotti. Formalizing Turing Machines. In *Logic, Language, Information and Computation - 19th International Workshop, WoLLIC 2012, Buenos Aires, Argentina*, volume 7456 of *Lecture Notes in Computer Science*, pages 1–25, 2012.

[9] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita Interactive Theorem Prover. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE-2011), Wroclaw, Poland*, volume 6803 of *LNCS*, 2011.

[10] Andrea Asperti, Wilmer Ricciotti, and Claudio Sacerdoti Coen. Matita tutorial. *Journal of Formalized Reasoning*, 7(2), 2014.

[11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009.

[12] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science*, 8(1), 2012.

[13] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Log.*, 9(1), 2007.

[14] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005.

[15] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA*, pages 3–15. ACM, 2008.

[16] G. Boolos, J.P. Burgess, and R.C. Jeffrey. *Computability and Logic, 5th edn.* Cambridge University Press, 2007.

[17] Bruno Buchberger. Certain decompositions of Gödel numbering and the semantics of programming languages. In *International Symposium on Theoretical Programming, 1972*, volume 5 of *Lecture Notes in Computer Science*, pages 152–171. Springer, 1974.

[18] Martin Davis. *Computability and Unsolvability.* Dover Publications, 1985.

[19] R. E. Stearns F. C. Hennie. Two-tape simulation of multi tape Turing machines. *Journal of ACM*, 13(4):533–546, 1966.

[20] Patrick C. Fischer. On Formalisms for Turing Machines. *J. ACM*, 12(4):570–580, 1965.

[21] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.

[22] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[23] Thomas Hales, Georges Gonthier, John Harrison, and Freek Wiedijk. A Special Issue on Formal Proof. *Notices of the American Mathematical Society*, 55, 2008.

[24] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transaction of the American Mathematical Society*, 117:285–306, 1965.

[25] F. C. Hennie. One-Tape, Off-Line Turing Machine Computations. *Information and Control*, 8(6):553–578, 1965.

[26] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[27] Gerwin Klein. Operating system verification – an overview. *Sadhana*, 34(1):27–69, 2009.

[28] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA*, pages 42–54, 2006.

[29] Michael Norrish. Mechanised Computability Theory. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011.

[30] Michael Sipser. *Introduction to the Theory of Computation*. PWS, 1996.

[31] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. of the London Math. Society*, 2(42):230–265, 1936.

[32] Peter van Emde Boas. Machine Models and Simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. North-Holland, Amsterdam, 1990.

[33] Stephanie Weirich and Benjamin Pierce (eds). Special Issue on the POPLmark Challenge. *J. Autom. Reasoning*, 49(3), 2012.

[34] Freek Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[35] Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising Turing Machines and Computability Theory in Isabelle/HOL. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.

[36] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. *Commun. ACM*, 54(12):123–131, 2011.